LABORATORY MANUAL

CE/CZ1106
Computer Organization and Architecture

Lab Experiment #2

Modular Programming using VisUAL

by
Asst. Prof. Mohamed M. Sabry Aly

SESSION 2020/2021
SEMESTER 1

SCHOOL OF COMPUTER SCIENCE AND ENGINEERING
NANYANG TECHNOLOGICAL UNIVERSITY

1. OBJECTIVES

1.1 Understand the concepts of modular programming and the use of subroutines for implementing program modules.

1.2 Understand how the stack is used to implement subroutine calls.

1.3 Understand the different techniques for passing parameters to a subroutine.

2. LABORATORY

This experiment is conducted at the **Hardware Lab 2** at
N4-01b-05 (Tel: 67905036).

3. EQUIPMENT

3.1 Hardware

- Personal Computer with any operating system.

3.2 Software / User Manuals

- VisUAL simulator software (https://salmanarif.bitbucket.io/visual/)
- Supported instructions in VisUAL

4. INTRODUCTION

Good programming techniques dictate that programs be **modular** for ease of debugging and testing. Each module or subroutine performs a concisely defined function, which can be called from a main program when needed. The ARM processor provides support for modular programming through. There are three general ways to pass parameters to a subroutine and they are:

- using registers
- using memory block
- using the stack

In this experiment, you will execute ARM assembly language programs that will illustrate how these three parameter-passing techniques can be implemented. For each example, a subroutine call **`Mean`** will be implemented to add the values in two 32-bit memory variables (**`Num1`** and **`Num2`**). The sum is then divided by 2 to obtain the mean value (remainder is ignored). The result of the 32-bit mean value is to be stored into another memory variable (**`Result`**). The memory variables **`Num1`**, **`Num2`** and **`Result`** are at addresses 0x100, 0x104 and 0x108 respective and they are only known to the calling program.

You will also observe in these examples, the two types of parameters that can be passed to a subroutine, namely passing **by value** and passing **by reference**. You will also be given an opportunity to modify a subroutine to ensure that it can be used in a **transparent** manner by a calling program. In other words, using this transparent subroutine will not cause any unintended side-effects in the calling program due to changes in the content of shared registers.

4.1 Notations used in this manual

`0x20` is used to specify a 32-bit hexadecimal (hex) value of 00000020_{16}.

5. EXPERIMENT

5.1 Passing parameters using registers

The first assembly language program (see Figure 1) shows how the values at two variables stored at addresses 0x100 and 0x104 (**Num1** and **Num2**) are passed to the subroutine **Mean** using registers. The result, passed out via a register, is then copied to memory variable (**Result**) at address 0x108.

```
;Calling Program
        ADR R0,Num1
        ADR R1,Num2
        ADR R2,Result       ; setup registers with values
        ADR SP, 0xFFFFFFFC  ; (a) initialize Stack Pointer (SP) to top of memory
        LDR R0,[R0]         ; (b) move value of Num1 stored at 0x100 into R0
        LDR R1,[R1]         ; (c) move value of Num2 stored at 0x104 into R
        BL Mean             ; (d) call subroutine Mean
        STR R1, [R2]        ; move result of mean to memory var at 0x108
        END                 ; Stop emulation

;Subroutine Mean
Mean    ADD  R1,R0,R1       ; add the two parameters in R0 and R1
        ASR  R1,R1,#1       ; divide-by-2 using arithmetic shift right by 1 bit
        MOV PC,LR           ; return to calling program
```

Figure 1 – The program for passing parameters by registers.

5.1.1 First run up the Visual emulator.

5.1.2 **Type in program** - Enter the mnemonics for the program given in Figure 1. In order to save time, you may want to leave out the comments (i.e. text after the ";" symbol) as they are not necessary for the execution of these instructions. You may want to use the "**Lab2-template**" as it has some memory locations initialized with useful values.

5.1.3 **Initializing the stack pointer (SP)** – Why do you think the stack pointer was initialized to start at the highest memory address 0xFFFFFFFC? (Hint: see the way the stack grows in Figure 2)

5.1.4 **Subroutine Call and the Stack** – Firstly, fill in the contents of the memory locations in Figure 2(a) before you start single stepping through the program. Once you have executed the **BL Mean** instruction, fill in the memory contents in Figure 2(b).

(a) What has been pushed to the stack? Draw where **SP** is pointing to in Figure 2(b).

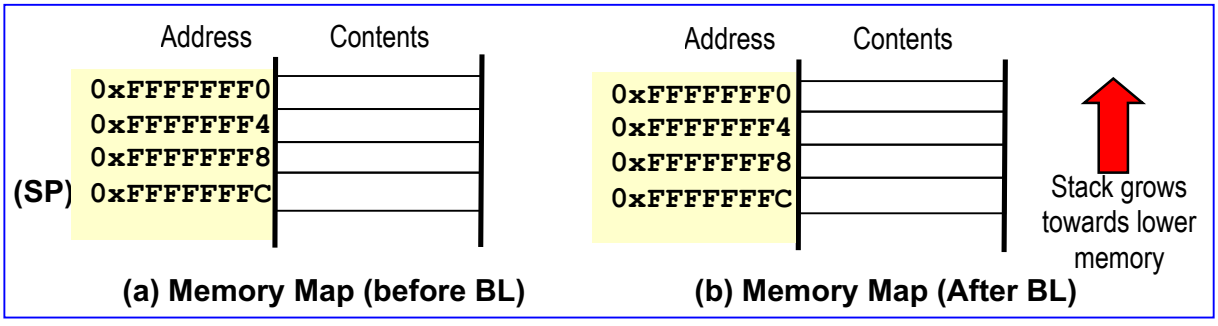

Figure 2 - Memory contents on the stack before and after the execution of the `BL Mean` instruction.

5.1.5 **Parameter passing** – In order to pass the parameters via the registers, the values of **`Num1`** and **`Num2`** stored in memory address 0x100 and 0x104 respectively, must be copied to the registers. Likewise, any result passed out by the subroutine via a register needs to be copied to the memory variable **`Result`** after returning from the subroutine.

(a) Which two registers are being used to pass the parameters to subroutine **`Mean`**?

(b) Which register is used to pass out the result to the calling program?

(c) Which other register is well known to return values from subroutine?

5.1.6 **Returning from subroutine** – In order to return to the calling program, the **`MOV PC,LR`** instruction makes use of link register. Using Figure 3, fill in the contents of the **PC** and **LR** before and immediate after the execution of the **`MOV PC,LR`** instruction.

(a) Has the stack has been affected in anyway by the execution of the **`MOV PC,LR`** instruction? If so, explain how and why the stack is affected in the manner observed.

(b) What has been copied into the **PC** after executing the **`MOV PC,LR`** instruction? With this **PC** content, which instruction do you think will be executed next?

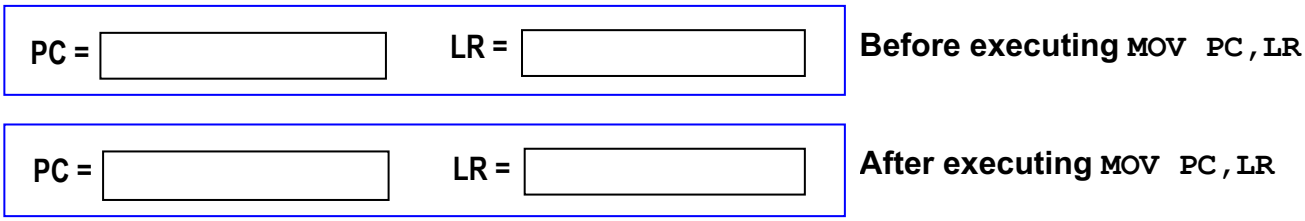

Figure 3 - Program counter and LR the `MOV PC,LR` instruction.

5.2 Passing parameters using memory block

Another method of passing parameters to a subroutine uses a parameter block. A parameter block is a set of contiguous memory locations containing the parameters. The parameters to be passed can be stored in this block of memory before calling the subroutine and subsequently retrieved from the same block whilst within the subroutine. Since memory is being used, the number of parameters passed can be numerous and not limited by the available registers. The program shown in Figure 4 illustrates how a memory block can be used to pass parameters to a subroutine.

```
;Calling Program
          ADR   SP,0XFFFFFFFC ; initialize Stack Pointer (SP) to top of memory
          MOV   R0,#0x100     ; (a) move start address of memory block into R0
          BL Mean             ; call subroutine Mean
          END                 ; End the program

;Subroutine Mean
Mean      LDR   R1,[R0]       ; (b) Load value in Num1 into R1
          LDR   R2,[R0,#4]    ; (c) Load Num2 in R2
          ADD   R1,R1,R2      ; Add R2 to R1
          ASR   R1,R1,#1      ; divide-by-2 using arithmetic shift right by 1 bit
          STR   R1,[R0,#8]    ; (d) move result in R1 into memory variable Result
          MOV   PC,LR         ; return to calling program
```

Figure 4 – The program for passing parameters using a memory block.

5.2.1 **Type in program** - Enter the mnemonics for the program given in Figure 4. In order to save time, you may want to leave out the comments.

5.2.2 **The memory block** – Using Figure 5(a), draw the contents of the memory block before the start of program execution.

Note: Click the **View memory contents** button on the **Tools** in VISUAL to see the contents in Data Memory.

5.2.3 **Base address of memory block** – In order for the subroutine to be able to retrieve parameters stored in the memory block, the base address (i.e. start address) of the memory block must be conveyed to the subroutine.

By observing instruction (a) in Figure 4, describe how the base address of the memory block is passed to the subroutine **`Mean`**.

5.2.4 **Retrieving parameters from memory block** – Based on the instructions (b) and (c) in Figure 4, what addressing mode is used to retrieve the values of **`Num1`** and **`Num2`** from the memory block? Use a different addressing mode to retrieve the values

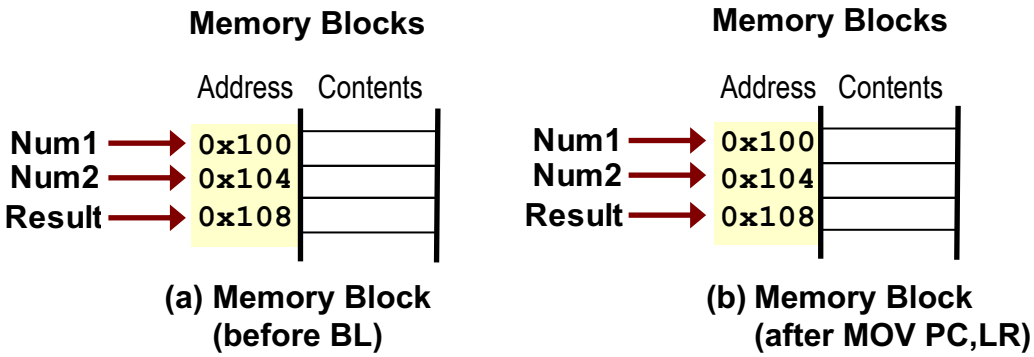

Figure 5 – The contents in the memory block (a) before calling the subroutine Mean and (b) after returning from the subroutine.

5.2.5 **Returning the result to the calling program** – By observing instruction (d) in Figure 4, describe how the result of the mean value computed within the subroutine **`Mean`** is returned to the calling program.

(a) Using Figure 5(b), draw the contents of the memory block immediately after execution the **`MOV PC,LR`** instruction.

(b) Has the correct mean value been updated into the memory variable **`Result`** in address 0x108?

With reference to the program in Figure 4, would you say the parameters were passed by value or passed by reference? (Hint: see section 5.3)

5.3 Passing parameters using the stack

The most general method of passing parameters to a subroutine is by using the stack. The parameters to be passed can be pushed onto the stack before calling the subroutine and subsequently retrieved from the stack whilst within the subroutine. Since a system stack is accessed via a stack pointer (**SP**), there is no need to pass the start address of a memory block to the subroutine. The number of parameters that can be passed is only limited by the maximum size of the system stack. The program shown in Figure 6 illustrates how the stack can be used to pass parameters to a subroutine.

In this example, the two numbers (**Num1** and **Num2**) whose mean is to be computed is passed by **value** (i.e. actual values stored in these memory variables are passed to the subroutine). In order to get the subroutine to directly update the result into memory variable **Result** at address 0x108, the variable **Result** is passed by **reference** (i.e. the address of the variable is passed to the subroutine).

```
;Calling Program
        ADR SP,0xFFFFFFFC   ; initialize Stack Pointer (SP) to top of memory
        MOV R0,#0x100       ;
        LDR R1,[R0],#4
        STR R1,[SP,#-4]!    ; (a) push value at memory variable Num1 to stack
        LDR R1,[R0],#4
        STR R1,[SP,#-4]!    ; (b) push value at memory variable Num2 to stack
        STR R0,[SP,#-4]!    ; (c) push address  at memory variable Result to stack
        BL Mean             ; call subroutine Mean
        ADD  SP,SP,#12      ; (h) remove the 3 parameters push to the stack earlier
        END                 ; Stop emulation

;Subroutine Mean
Mean    LDR  R0,[SP,#8]     ; (d) move value of Num1 that is on the stack into R0
        LDR  R1,[SP,#4]     ; (e) move value of Num2 that is on the stack to R1
        ADD  R0,R0,R1       ; Add R1 to R0
        ASR  R0,R0,#1       ; divide-by-2 using arithmetic shift right by 1 bit
        LDR  R2,[SP]        ; (f) move address of Result that is on the stack into R2
        STR  R0,[R2]        ; (g) move result in R0 into memory variable Result
        MOV PC,LR           ; return to calling program
```

Figure 6 – The program for passing parameters using the stack.

5.3.1 **Type in program** - Enter the mnemonics for the program given in Figure 6. In order to save time, you may want to leave out the comments.

5.3.2 **Parameters on the stack** – Reset the **PC** to that start of the program and single step through the program till you reach instruction (d) in Figure 6. Using Figure 7, fill in all the known items on the stack. Label these items with appropriate names such as **Num1**, **Num2**, and address of **Result**. Also indicate where **SP** is currently pointing to. Modify the caller to push all results, in their right sequence, to the stack in a single instruction.

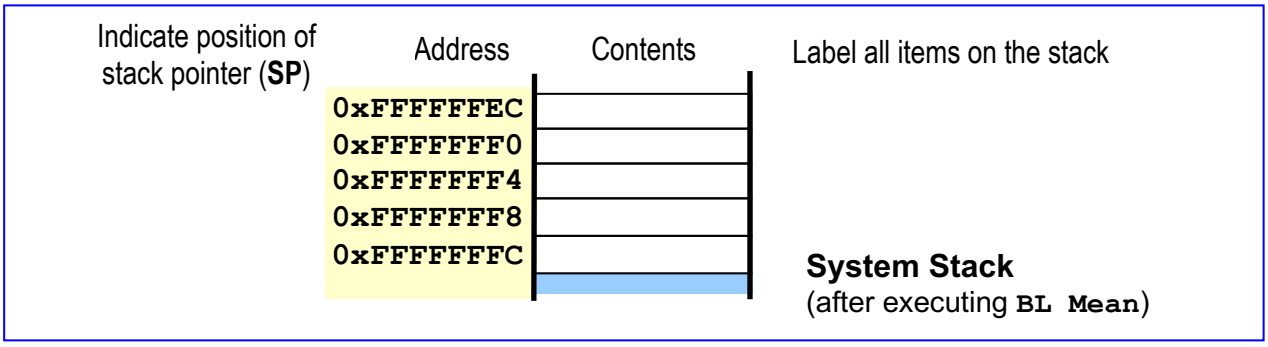

Figure 7 – The items on the system stack just after entering subroutine Mean.

5.3.3 **Assessing parameters on the stack** – Continue to step through instructions (d) to (g) in Figure 6 as you answer the following questions:

(a) What type of addressing mode is used to access the value of **`Num1`** and **`Num2`** that was pushed to the stack earlier in the calling program?

(b) Using the Figure 7 that you have filled in with the items on the stack, explain why the offsets values 8 and 4 were used to fetch the value of **`Num1`** and **`Num2`** on the stack respectively?

(c) Describe what instructions (f) and (g) do? What item is being assessed from the stack and how is this item used to return the result calling program? Can we use less regsiters than R0-R2?

5.3.4 **Removing parameters from the stack after subroutine call** – Identify which instruction in Figure 6 was used to perform the necessary housekeeping of getting rid of parameters that were pushed to the stack when they are no longer needed.

(a) Describe why this instruction is equivalent to the operation of popping off the 3 items that was pushed to the stack before the subroutine?

(b) What would happen if parameters were repeatedly pushed to the stack before subroutine calls but not removed from the stack after returning from the subroutine?

5.1 Writing a transparent subroutine

An important characteristic of a subroutine is that it should be **transparent**. In other words, it does not affect the proper execution of a program after it has been called within the program. One way of achieving transparency is to ensure that before one exits the subroutine, all registers used within the subroutine (except the register used to pass out the result) are restored with the original contents they had just before entering the subroutine.

The **`STMFD`** and **`LDMFD`** instructions can be used to efficiently save all used registers to the stack on entry, and restore them before returning from the subroutine.

Modify the subroutine **`Mean`** in Figure 6 to save away registers **R0, R1, and R2** to the stack on entry into the subroutine and then restore their original values from the stack just before executing the **`MOV PC,LR`** instruction. Do note that additional items have been pushed to the stack when you carry out this operation. Think carefully about what **new offset values** (i.e. instructions (d), (e) and (f) in Figure

6) are required to correctly access the stack-based parameters **`Num1`**, **`Num2`** and address of **`Result`** as there are now additional items on the stack.

Step through your modified program to verify that your subroutine still works correctly and the values of **R0** and **R1** and **R2** before and after your subroutine call remains unchanged.

Note: The remaining sections are optional. You should proceed to complete it if you have already completed section 5.1 to 5.3.

5.2 Programming Exercise

A C string can be found starting at address 0x100. The end of a C string is terminated with a character of value of 0x000. Using the template given to you in the VIPAS project file "**Lab2-5_5-template**", complete the subroutine **`strlen`** that will return the length of the string in register **R0** to the calling program. The length of your string should **include** the count for the terminator character 0x000. The start address of the string is passed to **`strlen`** using the stack.

The template for "**Lab2-5_5-template**" is given in Figure 8 but use the provided VISUAL project file as the data memory has been filled with appropriate values. If you have written **`strlen`** correctly, it should return the value of 21 or 0x15.

```
;Calling Program
        ADR   SP,0xFFFFFFFC  ; initialize Stack Pointer (SP) to top of memory
        MOV   R0,#0x100      ; Fill R0 with start address
        STMFD SP!,{R0}       ; push start address of string to stack
        BL Strlen            ; call subroutine Mean
        ?                    ; perform appropriate housekeeping of the stack
        END                  ; Terminate

;Subroutine Strlen
Strlen  ??                   ; save registers used in Strlen subroutine to stack
        ??
        :                    ; complete the subroutine Strlen
        ??
```

Figure 8 – The program template for the Strlen programming exercise.

6. LOG BOOK

Record your results, observations and analysis in section 5 into your **log book**. You may find some of this information helpful in the open book quiz that you will be taking before the end of this laboratory session

7. REFERENCES

7.1 CE1106/CZ1106 Lecture Notes on Modular Programming by Asst. Prof. Mohamed M. Sabry Aly (2020)

7.2 The VISUAL User Guide